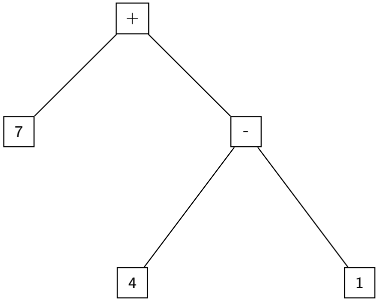
Linear Ordered Attribute Grammars without fake dependency selection

L. Thomas van Binsbergen, Jeroen Bransen, Atze Dijkstra
Utrecht University

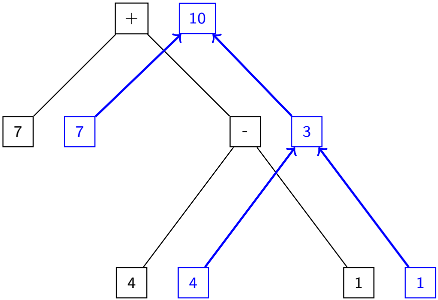
TFP 2014

Universiteit Utrecht

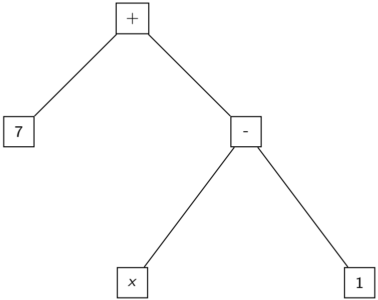
Evaluating Expressions



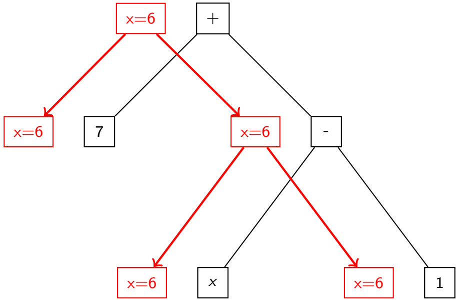
Evaluating Expressions



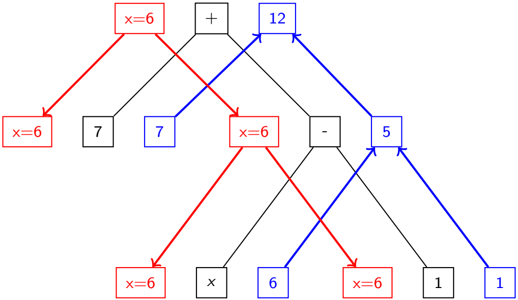
Evaluating Expressions



Evaluating Expressions

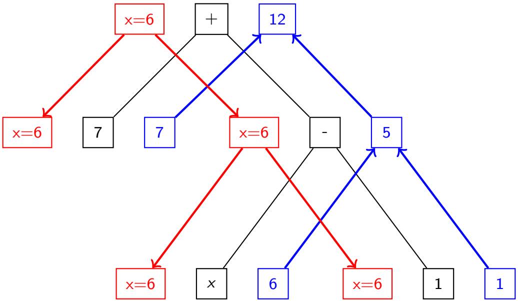


Evaluating Expressions



Evaluating Expressions using Attribute Grammars

- Attribute Grammars extend trees with attributes.
- Every node N represents one or more functions, that:
  - Receive a subset of the inherited attributes of N.
  - Produce a subset of the synthesized attributes of N.
- Attribute Grammars form a DSL for tree-based computations.



Universiteit Utrecht

Scheduling Multiple Computations

- Easily define multiple computations on the same tree.
- Leave worrying about the evaluation order to some compiler:
  - Host compiler: generating catamorphisms.
  - AG compiler: finding a static evaluation order.

Scheduling Multiple Computations

- To find a static evaluation order, we need to:
  - Find an interface for every nonterminal.
  - Show how every production implements it.
- An AG for which this is possible is linear ordered (LOAG).
- Decision problem:
  "Is there an assignment $i \to s$ or $s \to i$, for every pair (i, s) of every nonterminal X with $i \in inh(X)$ and $s \in syn(X)$, such that there are no dependency cycles?"
- Membership decision is NP-Hard.

Linear Ordered AGs

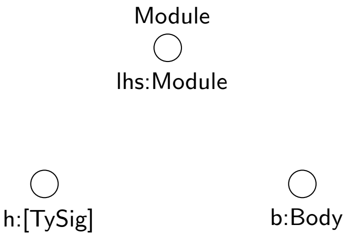
- Largest class that allows static evaluation schedules.
- Assumed to generate efficient code.
- Only algorithms for subclasses exist.
- Paper describes an algorithm to schedule LOAGs:
  - Exponential in theory, efficient in practice.
  - Compiles the UHC without assistance from the programmer.

Scheduling the UHC

- UHC is partly generated from of a large number of AGs.
- The "main AG" is very large indeed:
  - 30 nonterminals
  - 134 productions
  - 1332 attributes (44.4 per nonterminal!)
  - 9766 dependencies
- The main AG is written as a *linear ordered* AG.
- Kastens Algorithm does not recognise it.
- More then 20 fake dependencies required to help scheduling.

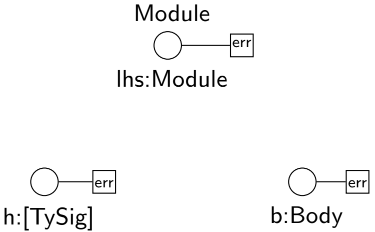
iModule: Types

```
module BinIntTrees
   flatten :: Tree → [Int]
   sum :: Tree → Int
where
   data Tree = Bin Tree Tree
             | Leaf Int
   flatten (Leaf i) = [i]
   flatten (Bin l r) = flatten l ++ flatten r
   ...
```



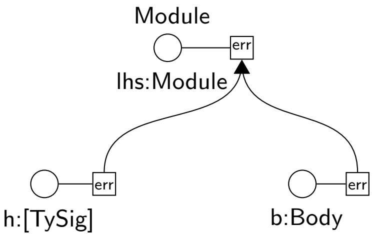
iModule: Types

```
module BinIntTrees
    flatten :: Tree → [Int]
    sum :: Tree → Int
where
    data Tree = Bin Tree Tree
              | Leaf Int
    flatten (Leaf i)  = [i]
    flatten (Bin l r) = flatten l ++ flatten r
    ...
```



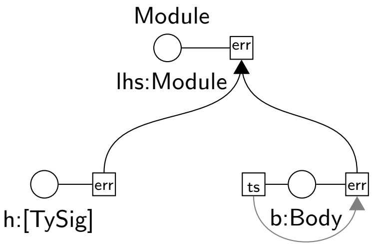
iModule: Types

```
module BinIntTrees
   flatten :: Tree → [Int]
   sum :: Tree → Int
where
   data Tree = Bin Tree Tree
             | Leaf Int
   flatten (Leaf i)  = [i]
   flatten (Bin l r) = flatten l ++ flatten r
   ...
```



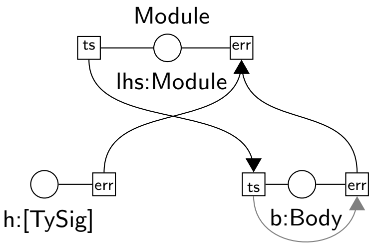
iModule: Types

module *BinIntTrees*
  flatten :: *Tree* → [*Int*]
  sum :: *Tree* → *Int*
where
  data *Tree* = *Bin Tree Tree*
         | *Leaf Int*
  flatten (*Leaf i*) = [*i*]
  flatten (*Bin l r*) = *flatten l* ++ *flatten r*
  ...



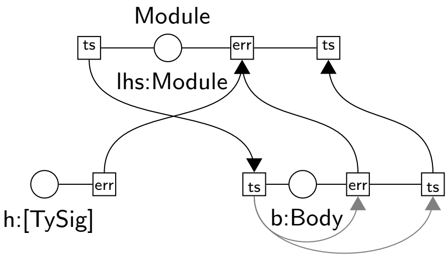
iModule: Types

$$
\begin{array}{l}
\textbf{module}\ \mathit{BinIntTrees} \\
\quad \mathit{flatten} :: \mathit{Tree} \rightarrow [\mathit{Int}] \\
\quad \mathit{sum} :: \mathit{Tree} \rightarrow \mathit{Int} \\
\textbf{where} \\
\quad \textbf{data}\ \mathit{Tree} = \mathit{Bin}\ \mathit{Tree}\ \mathit{Tree} \\
\qquad\qquad\quad\ \mid\ \mathit{Leaf}\ \mathit{Int} \\
\quad \mathit{flatten}\ (\mathit{Leaf}\ i)\ \ = [i] \\
\quad \mathit{flatten}\ (\mathit{Bin}\ l\ r) = \mathit{flatten}\ l \mathbin{+\!\!+} \mathit{flatten}\ r \\
\quad \ldots
\end{array}
$$



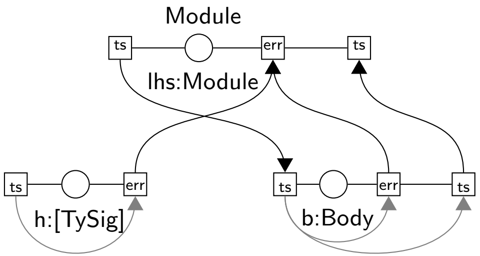
iModule: Types

$$
\begin{array}{l}
\textbf{module}\ \mathit{BinIntTrees} \\
\quad \mathit{flatten} :: \mathit{Tree} \to [\mathit{Int}] \\
\quad \mathit{sum} :: \mathit{Tree} \to \mathit{Int} \\
\textbf{where} \\
\quad \textbf{data}\ \mathit{Tree} = \mathit{Bin}\ \mathit{Tree}\ \mathit{Tree} \\
\qquad\qquad\qquad \mid\ \mathit{Leaf}\ \mathit{Int} \\
\quad \mathit{flatten}\ (\mathit{Leaf}\ i) \ = [i] \\
\quad \mathit{flatten}\ (\mathit{Bin}\ l\ r) = \mathit{flatten}\ l +\!\!+\ \mathit{flatten}\ r \\
\quad \ldots
\end{array}
$$



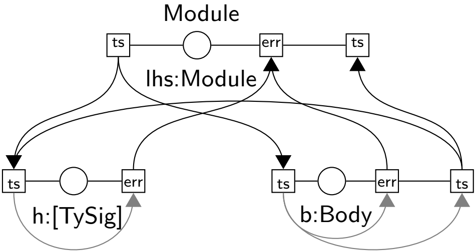
iModule: Types

```
module BinIntTrees
   flatten :: Tree → [Int]
   sum :: Tree → Int
where
   data Tree = Bin Tree Tree
             | Leaf Int
   flatten (Leaf i)  = [i]
   flatten (Bin l r) = flatten l ++ flatten r
   ...
```



iModule: Types

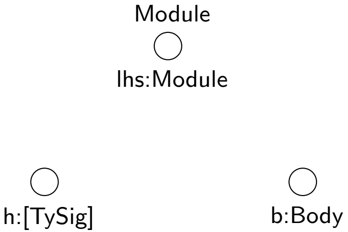
$$
\begin{array}{l}
\textbf{module } \mathit{BinIntTrees} \\
\quad \mathit{flatten} :: \mathit{Tree} \rightarrow [\mathit{Int}] \\
\quad \mathit{sum} :: \mathit{Tree} \rightarrow \mathit{Int} \\
\textbf{where} \\
\quad \textbf{data } \mathit{Tree} = \mathit{Bin}\ \mathit{Tree}\ \mathit{Tree} \\
\qquad\qquad\quad\ \mid \ \mathit{Leaf}\ \mathit{Int} \\
\quad \mathit{flatten}\ (\mathit{Leaf}\ i) \ = [i] \\
\quad \mathit{flatten}\ (\mathit{Bin}\ l\ r) = \mathit{flatten}\ l +\!\!+ \mathit{flatten}\ r \\
\quad \ldots
\end{array}
$$



Universiteit Utrecht

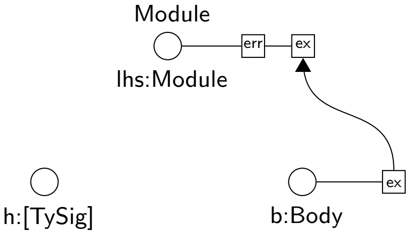
iModule: Exports

```
module BinIntTrees
   flatten :: Tree → [Int]
   sum :: Tree → Int
where
   data Tree = Bin Tree Tree
             | Leaf Int
   flatten (Leaf i)  = [i]
   flatten (Bin l r) = flatten l ++ flatten r
   ...
```



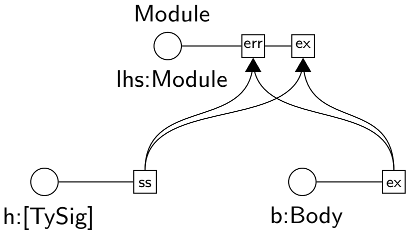
iModule: Exports

```
module BinIntTrees
   flatten :: Tree → [Int]
   sum :: Tree → Int
where
   data Tree = Bin Tree Tree
             | Leaf Int
   flatten (Leaf i)  = [i]
   flatten (Bin l r) = flatten l ++ flatten r
   ...
```

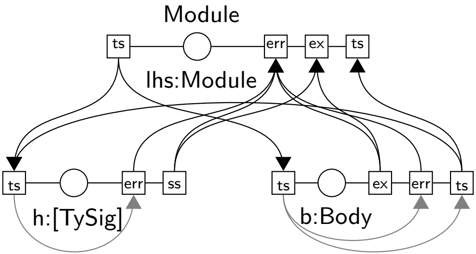


iModule: Exports

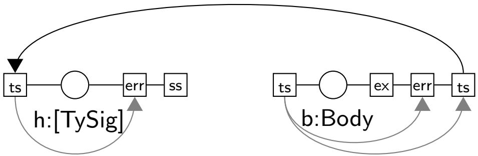
module *BinIntTrees*
    flatten :: *Tree* $\rightarrow$ [*Int*]
    sum :: *Tree* $\rightarrow$ *Int*
where
    data *Tree* = *Bin Tree Tree*
            | *Leaf Int*
    flatten (*Leaf i*) = [*i*]
    flatten (*Bin l r*) = *flatten l* ++ *flatten r*
    ...



iModule Combined

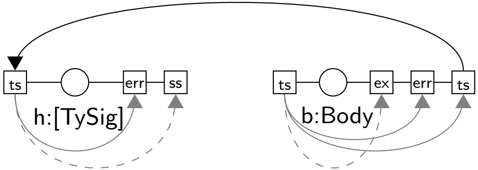


iModule Combined



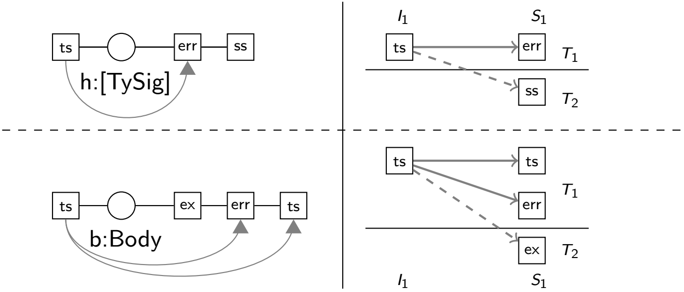
"Is there an assignment $i \rightarrow s$ or $s \rightarrow i$, for every pair (i, s) of every nonterminal X with $i \in inh(X)$ and $s \in syn(X)$, such that there are no dependency cycles?"

iModule Combined



"Is there an assignment $i \rightarrow s$ or $s \rightarrow i$, for every pair (i, s) of every nonterminal X with $i \in inh(X)$ and $s \in syn(X)$, such that there are no dependency cycles?"

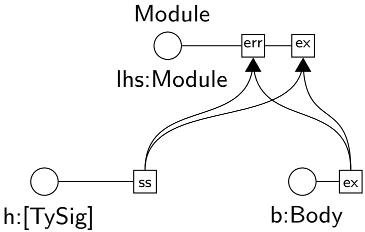
Kastens Algorithm



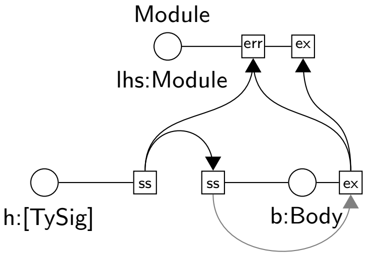
"Say $a \rightarrow b$ if a is at the i'th and b at the $(i-1)$'th position in the partial order implied by the dependencies."

Universiteit Utrecht

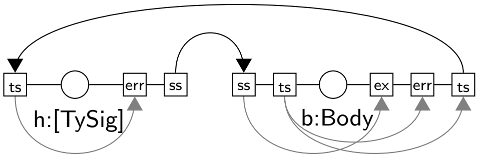
iModule: Exports Alternative



iModule: Exports Alternative

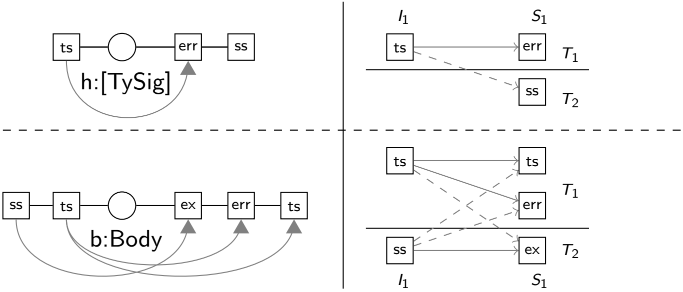


iModule Combined Alternative



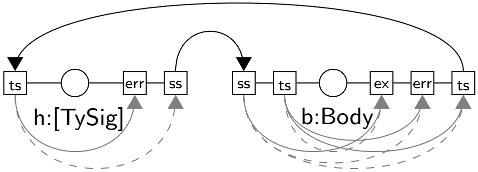
"Is there an assignment $i \to s$ or $s \to i$, for every pair (i, s) of every nonterminal X with $i \in inh(X)$ and $s \in syn(X)$, such that there are no dependency cycles?"

Kastens Algorithm

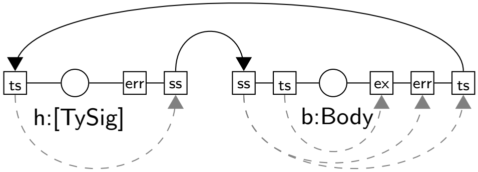


"Say $a \to b$ if a is at the i'th and b at the $(i-1)$'th position in the partial order implied by the dependencies."

Applying Kastens Algorithm



Applying Kastens Algorithm



Approach

1. Make all missing assignments optimistically.
2. When a cycle without optimistic assignments is encountered: AG $\notin$ LOAG.
3. When a cycle c with optimistic assignments is encountered:
   3.1 Select an optimistic assignment, swap it and recurse (goto 1).
      - The assignment is not considered optimistic anymore.
      - If AG $\notin$ LOAG is returned: swap other assignment and recurse.
      - If no more optimistic assignments in c: AG $\notin$ LOAG.
      - Otherwise: AG $\in$ LOAG.
4. AG $\in$ LOAG.

Results

- We can now compile the UHC without fake dependencies.
- In comparable time, 25 sec vs 35 sec.
- 10 corrections and no backtracking required!
- Most time is spent propagation dependencies.

Conclusions and Contributions

- We have given a decision procedure for the class of LOAGs.
- Exponential algorithm in theory, efficient in practice.
- Backtracking should be rare.

Future Work

LOAG algorithm

- We built a fast LOAG algorithm using SAT-solving (< 10 sec).
- We wish to show it can be optimised for specific needs.
- Perhaps the approach can be generalised for other problems.

Code efficiency

- Formalise the costs of schedules by fixing an execution model.
- Possibly by developing a specialised virtual machine for AGs.
- Compare existing algorithms and the schedules they produce.
- Extend the algorithm(s) with user defined optimisations.

Attribute Grammars

Why AGs are used

- AGs form a DSL for tree-based computations.
  - Semantics.
  - Static analyses.
- Declarative programming in imperative settings.
- High level of abstraction and concern separation.
- UUAGC generates the boilerplate code we need.

Why AGs aren't used

- Generated code is often not optimal.
- Finding a static evaluation order is hard:
  - Scheduling UHC requires manual assistance.

Universiteit Utrecht

Questions

How fast discovers the algorithm that an AG $\notin$ LOAG?

- Very fast for Circular AGs.
- Problematic on Absolutely Non-Circular AGs (time ???).

Questions

Why is LOAG preferred over ANCAG?

- Static schedules. (simple evaluators, optimisations)
- Context-free evaluation.

To program with AGs is

1. Specifying the Abstract Syntax Tree.
2. Add attributes to nonterminals.
3. Specifying the 'arrows'.

AG Syntax

data Module | Module *h* : [**TySig**] *b* : **Body**
...
attr Module
  syn *err* : `Bool`
  ...
sem Module | Module
    $lhs.err = @b.err \lor @h.err$

Design Choices

- Which AG compiler to use?
- Which host language to use?
- Which code generation procedure to use?
- Which (data)types to use for the attributes?
- How to compute all attributes with incoming arrows?

Design Choices

- Which AG compiler to use? UUAGC.
- Which host language to use?
- Which code generation procedure to use?
- Which (data)types to use for the attributes?
- How to compute all attributes with incoming arrows?

Design Choices

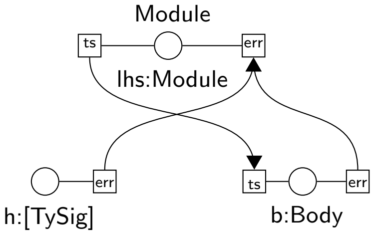
- Which AG compiler to use? UUAGC.
- Which host language to use? Haskell.
- Which code generation procedure to use?
- Which (data)types to use for the attributes?
- How to compute all attributes with incoming arrows?

Design Choices

- Which AG compiler to use? UUAGC.
- Which host language to use? Haskell.
- Which code generation procedure to use? LOAG.
- Which (data)types to use for the attributes?
- How to compute all attributes with incoming arrows?

Using the UUAGC

- Copy rules generate ‘logistics’.
- Use rules combine 0,1,2,. . . attributes.



- Local attributes act like new terminals.
- Higher-order attributes act like new nonterminals.

Universiteit Utrecht

Code Generation Procedures

- Circular AGs: Relying on Haskell's laziness.
- Absolutely Non-Circular AGs: Kennedy-Warren algorithm.
- Ordered AGs: Kastens algorithm.

Code Generation Procedures

- Circular AGs: Relying on Haskell's laziness.
- Absolutely Non-Circular AGs: Kennedy-Warren algorithm.
- Linear Ordered AGs.
- Ordered AGs: Kastens algorithm.